9

Filas de Prioridade

" Há duas maneiras de construir um projecto de software: uma maneira de fazer isso deve ser tão simples que obviamente não deixa deficiências e, a outra é torna-la tão complicada que não se percebe as evidentes deficiências. O primeiro método é o mais fácil "

- **Car Hoare** -

Sumário:

Estrutura de Dados e Algoritmos

9.1 - Conceitos

Existem algumas aplicações cujo funcionamento baseia-se na propriedade de executar as operações com o elemento que tiver o maior grau de prioridade. Para essas aplicações cada elemento possui um campo onde para armazenar esse grau.

Vejamos um exemplo de utilização muito comum. Os sistemas de atendimento a paciêntes num hospital, têm um campo onde os "socorristas" determinam o estado do paciênte. Em função desse estado (grau de prioridade) os paciêntes que estiverem em pior situação serão atendidos em primeiro lugar.

9.2 - Definições e Propriedades

Entendemos por **Fila de prioridade** um conjunto finito de elementos que está associado à um grau de prioridade. Esse grau é geralmente definido por um valor númerico que está armazenado em cada elemento desse conjunto

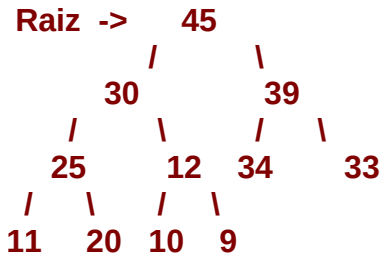
As Filas de prioridade podem ser implementadas em vectores não ordenados, ordenadas e em Heap's. Estudaremos nessas notas a sua implementação num heap por ser a estrutura mais eficiênte e adequada para representar esse conjunto.

Um **Heap** é uma àrvore de busca binária de profundidade h que possui as seguintes propriedades:

- Todas as folhas estão no nível h ou h-1.
- Até ao nível h-1, todos os átomos têm necessáriamente dois filhos.
- As folhas estão encostadas à esquerda.

Para além disso, para cada átomo,o grau de prioridade desse átomo é menor do que o grau de prioridade dos seus filhos (Heap no sentido maximal, denominado por Max-Heap) ou o grau de prioridade desse átomo é maior do que o grau de prioridade dos seus filhos (Heap no sentido minimal, denominado por Mini-Heap). Esta propriedade de extrema importância e será chamada de **regra pai-filho**.

Vejamos em seguida, um exemplo ilustrativo de um Max-Heap

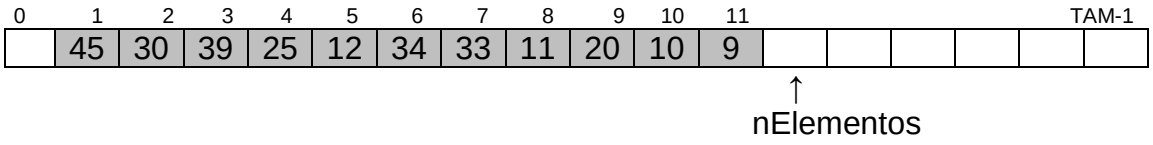


Então, podemos concluir que um heap é uma árvore binária de profundidade h quase completa que satisfaz a regra pai-filho.

Para facilitar a implementação dos algoritmos vamos considerar que os elementos estão armazendos nas posições 1,2,...m e não 0,1,... como é o padrão na linguagem C.

O processo de armazenamento num heap consiste em inserir os elementos nível por nível da esquerda para a direita. Vejamos um exemplo ilustractivo com a àrvore descrita anteriormente.

Inicialmente armazenamos a raiz, o número 45 no elemento v[1]. Em seguida, armazenamos o filho esquerdo da raiz, o número 30 no elemento v[2] e o filho direito da raiz, o número 39 no elemento v[3]. Agora vamos armazenar nas próximas posições os filhos esquerdo e direito do número 30. Esses filhos serão armazenados nos elementos v[4] e v[5]. Como exercicio continue o processo de armazenamento e veá que teremos o seguinte vector:



Observamos que para um determinado índice i um índice do vector maior do que zero, as seguintes propriedades são verdadeiras:

- O filho esquerdo de i está na posição 2i
- O filho direiro de i está na posição 2i + 1

e para i > 1

- O pai de i está na posição i/2.

Para além dessa propriedade, este processo mostra que todas as folhas (elementos sem filhos) estão armazenada nas posições i, tais que:

i >= nElementos/2

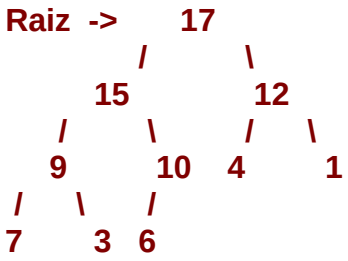
9.3 - Estrutura de Dados

Para desenvolvermos este estudo, vamos utilizar um vector de registros, denominado por H com MAX elementos do tipo THeap. Cada elemento desse vector possui entre outros campos, um campo denominado por prioridade do tipo inteiro. Associado a esse vector, temos uma variável do tipo inteiro, denominada por nElementos que dar-nos-á o número de elementos inseridos.

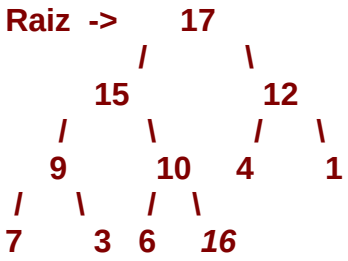
9.4 - Implementação das Operações

Deixaremos como exercício as implementações para inicializar um Heap e para verificar se o Heap está cheio. Contudo, adoptamos com o principio de heap está vazio o facto do número de elementos inseridos ser igual a zeros.

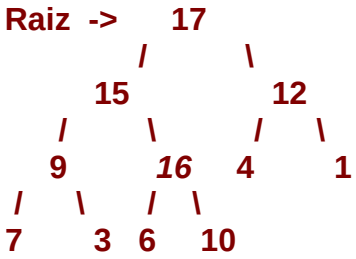
A operação de inserção de um elemento é feita no fim da fila. Para garantir a existência de um Max-Heap, o elemento inserido deve "subir" na estrutura da fila até que a regra pai-filho esteja satisfeita. Vejamos um exemplo ilustractivo: Dado o Heap.



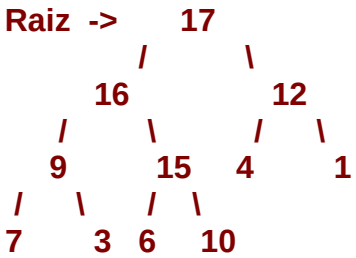
Vamos inserir o número 16 no fim da árvore (16 é o grau de prioridade).



Como 16 é maior do que 10, a regra pai-filho foi violada. Para restabelecê-la, trocamos o conteudo dos dois àtomos, obtendo:



Mas 16 é maior do que 15, isso quer dizer que regra pai-filho não foi restabelecida. Para restabelecê-la, vamos proceder mais uma vês a troca do conteúdo dos dois átomos, obtendo:



Agora, 16 é menor do que 17, a regra pai-filho foi restabelecida e como consequência, o processo de "subir" na árvore termina.

Estamos em condições de escrever uma função que implementa esta operação.

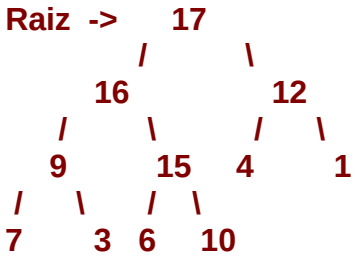
```
/*-------------------------------------------------------------------------------------------
Objectivo: Inserir  um elemento no fim do Max-Heap e reconstituir a árvore
Parâmetro Entrada: Heap e o  elemento a inserir
Parâmetro de Saída: Heap actualizado
-------------------------------------------------------------------------------------------*/
void  inserirHeap (THeap  *H[ ], Titem  x)
{
   int pai, filho;
   boolean sobe;
   H->nElementos ++;
   filho = H.nElementos;
   H->item[filho].chave = x;
   pai = filho/2;
   sobre = TRUE;
   while (( pai > 0) && (sobe))
      {
         if ( H->[pai].prioridade < x.prioridade )
           {
              troca(&H[pai],&H[filho];
              filho = pai;
              pai = pai/2;
            }
         else
             sobre = FALSE;
      }
}
```

A operação para remover um elemento é muito simples, ela consiste em retirar o elemento que está na primeira posição do vector, que para o nosso caso é o elemento que está na posição de índide um. Veja o algoritmo de heap vazio. Esse é o elemento que tem o maior grau de prioridade. Com essa operação, perdemos o conceito de Heap. Para restitui-lo, devemos colocar o último elemento da lista na primeira posição. Mas, essa movimentação tem duas consequências. Se o elemento movimentado tiver um grau de prioridade maior do que os seus filhos as propriedades do Max-Heap foram preservadas e não temos nada à fazer. Mas se isso não acontece, teremos de restaurar a árvore de tal forma que as propriedades do Max-Heap sejam garantidas.

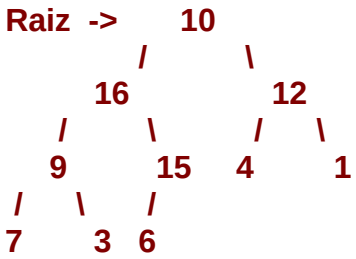
```
/*-------------------------------------------------------------------------------------------
Objectivo: Remover o elemento da raiz do Heap e reconstituir a árvore
Parâmetro Entrada: Heap e o número elementos inseridos
Parâmetro saída: Heap actualizado e o elemento removido
-------------------------------------------------------------------------------------------*/
```

```
void  removerHeap (THeap *H, Titem *x)
{
   *x = H->item[1];
    H[1] = H->item[A.nElementos];
    H->nElementos--;
    RestaurarHeap(1, &H);
}
```

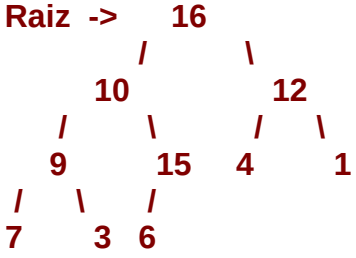
O processo de restauro, denominada por bubbling-up, consiste em "baixar" o elemento que está numa determinada posição na árvore até encontrar um nível onde a regra pai-filho esteja satisfeita. Essa operação consiste em trocar o conteúdo do pai pelo conteúdo do filho que tem o maior grau de prioridade. Repetir esse processo enquanto existirem trocas ou filhos. Vejamos um exemplo ilustractivo: Dado o Heap:



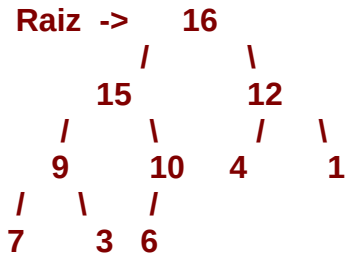
Vamos remover a raiz, o átomo com o grau de prioridade 17 e colocamos no seu lugar o último átomo da árvore.



Como 10 não é maior do que o grau dos seus filhos, a regra pai-filho foi violada. Para restabelecê-la, trocamos o conteudo do pai com o filho de maior prioridade obtendo:



Mas 10 não é maior do que o grau dos seus filho direito. Isso quer dizer que regra pai-filho não foi restabelecida. Devemos proceder novamente a troca desse átomo com o filho de maior prioridade obtendo:



Como regra pai-filho foi restabelecida e o processo de reorganização termina.

Apresentamos em seguida, uma função que implementa essa operação.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Restaurar o Max-Heap
Parâmetro Entrada:  Posição restauro e Estrutura do Heap
Parâmetro de Saída: Estrutura de max-Heap restaurada
-------------------------------------------------------------------------------------------------*/
void restaurarHeap (int i, THeap *H )
{
  while (i < H->nElementos/2)
   {
      int filhoesq = 2*i , filhodir = 2*i + 1, maiorfilho;
      if (filhodir <= T->nElementos )                          // se tem filho direito
          {
            if  (H->item[filhoesq].prioridade < H->item[filhodir].prioridade)
                maiorfilho = filhodir;
            else
                maiorfilho = filhoesq;
          }
       else
           maiorfilho = filhoesq;

       if (H->item[i].prioridade >= H->item[maiorfilho].prioridade)  break;
       troca (&H[i], &H[maiorfilho]);
       i = maiorfilho;                                          // desce
   }
}
```

9.5 - Ordenação por Árvore

Estamos em condições de estudar um método de ordenação por árvore, denominado por **(Heap Sort)** , proposto em 1964 pelos matemáticos Robert W. Floyd e J.W.J Williams.

Nestas notas, iremos debruçar apenas sobre o funcionamento do Max-Heap. Em termos gerais, o algoritmo que implementa esse método possui a seguinte descrição:

Inicialmente o vector contêm os dados do Max-Heap. Suponhamos sem perda da generalidade que esse Max-Heap possui n elementos.

Primeira iteração, removemos o primeiro elemento do heap, o elemento com a maior prioridade e procedemos a reorganização do heap. Como a n-ésima posição do vector está livre podemos aproveita-la para armazenar o elemento removido.

Segunda iteração, removemos o segundo elemento de maior prioridade do heap e armazenamos esse elemento na (n-1)-ésima posição do vector. Lembre-se que ao reorganizarmos o vector o elemento de maior prioridade é armazenado na primeira posição do heap.

O processo termina quando o Max-Heap for unitário. Nessa altura, o elemento com a menor prioridade encontra-se na primeira posição do heap.

Se percorrermos o vector do primeiro ao último elemento inserido, teremos um conjunto de dados ordenados por grau de prioridade.

Com base nesta descrição, estamos em condições de implementar o seguinte procedimento.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Ordenar os elementos do  Max-Heap
Parâmetro Entrada: Estrutura de um Max-Heap
Parâmetro de saída: Vector ordenado em ordem crescente
-------------------------------------------------------------------------------------------------------*/
void  heapSort (THeap *H)
{
   TItem x;
   for ( int i = H->nElementos; i >= 2; i--)
     {
        troca (&H[1], &H[i]);
        T->nElementos--;
        restaurarHeap (1, &T );
      }
}
```

Mas, para executarmos este procedimento necessitamos de construir o heap no vector. Esta acção de construção deve ser feita por uma função que irá invocar o heap Sort, dando a garantia que estão garantidas a organização no max-heap no vector e o número de elementos inseridos, que serão passados como parâmetros.

Suponhamos que o vector está vazio. Para esse caso, o procedimento consiste em ler um conjunto finito de dados que termina com um sentinela de fim de leitura e inserir esses dados no vector respeitando as regras do Max-Heap. A função que descrevemos a seguir realiza essa tarefa.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Inserir os elementos no vector de forma a constituir um  Max-heap
Parâmetro Entrada: Vector
Parâmetro de Saída: Estrutura de max-heap
Valor de Retorno: Número de elementos inseridos
-------------------------------------------------------------------------------------------------------*/
int ControiHeap (THeap *H )
{
  inicializarHeap (&H);
  THeap dado = LerDados();
  while (dado.chave != SENTINELA )
     {
        if ( cheioHeap(H) )
          {
            printf("\N Erro: Vector está cheio");
            return 0;
          }
        inserirHeap (&H dado);
        dado = LerDados();
     }
   return H->nElementos;
 }
```

Agora, vamos supor que o vector que contêm o Max-Heap não estiver vazio. A função carregarHeap() consiste em organizar os elementos do vector de forma a preservar as propriedades do Max-Heap. Mas como todos as folhas estão em posições iguais ou superiores ao nElementos/2 então essa restauração deverá ser feita no intervalo {0.. nElementos/2} e consiste na seguinte função.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Restaura os elementos de forma a construir um max-heap
Parâmetro Entrada: vector
Parâmetro de Saída: Estrutura de max-heap
-------------------------------------------------------------------------------------------------------*/
int carregarHeap (THeap *H)
{
   for (int i = H->nElementos/2 ; i >= 1; i--)
        restaurarHeap (i,&T);
    return H->nElementos;
}
```

9.6 - Exercícios

9.6.1- Desenvolva uma função recursiva que recebe como argumento um índice e o Heap, restaurar o Heap no sentido maximal.

9.6.2- Desenvolva uma função iterativa que recebe como argumento a posição de um elemento num Heap. Verificar se esse elemento satisfaz a regra pai-filho.

9.6.3- Desenvolva uma função recursiva que recebe como argumento uma estrutura de Max-Heap e um determinado elemento. "Subir" o elemento na árvore até encontrar um nível onde a regra pai-filho esteja satisfeita. Sempre que fizer esse processo deve trocar os pai pelo filho.

9.6.5- Desenvolva uma função recursiva que recebe como argumento uma estrutura de Max-Heap, um determinado índice nessa estrutura e um determinado valor. Subir na estrutura da árvore até encontrar uma posição que satisfaça a regra pai-filho.

9.6.7- Desenvolva uma função iterativa que recebe como argumento uma estrutura de Max-Heap e devolve essa estrutura ordenada na ordem crescente:

33 32 28 31 26 29 25 30 27
33 32 28 31 29 26 25 30 27

9.6.8- Dado o seguinte conjunto de dados:

18 25 41 34 14 10 52 50 58

Determinar o Heap obtido pela aplicação do algoritmo de construção.